Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric

Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh
Cornell University, Ithaca, NY, USA
{*dyd2,dl575,gjm76,ss868,gs272*}*@cornell.edu*

***Abstract*—This paper proposes FlexCore, a hybrid processor architecture where an on-chip reconfigurable fabric (FPGA) is tightly coupled with the main processing core. FlexCore provides an efficient platform that can support a broad range of run-time monitoring and bookkeeping techniques. Unlike using custom hardware, which is more efficient but often extremely difficult and expensive to incorporate into a modern microprocessor, the FlexCore architecture allows parallel monitoring and bookkeeping functions to be dynamically added to the processing core and adapt to application needs even after the chip has been fabricated. At the same time, FlexCore is far more efficient than software implementations because its fine-grained reconfigurable architecture closely matches bit-level operations of typical monitoring schemes and allows monitoring schemes to operate in parallel to the monitored core. In fact, our experimental results show that monitoring on FlexCore can almost match the performance of full ASIC implementations. To evaluate the FlexCore architecture, we implemented an RTL prototype along with several extensions including uninitialized memory read checking, dynamic information flow tracking, array bound checking, and soft error checking. The prototypes demonstrate that the architecture can support a range of monitoring extensions with different characteristics in an efficient manner. FlexCore takes moderate silicon area and results in far better performance and energy efficiency than software.**

I. Introduction

As we expand the use of computing devices, capabilities beyond raw performance such as support for security, reliability, and programmability are becoming increasingly important. Run-time monitoring of program execution at an instruction granularity presents an effective way to ensure a wide range of security and reliability properties and enhance the programmability of a system. As an example, Dynamic Information Flow Tracking (DIFT) is a recently proposed security technique that tracks and restricts the use of untrusted I/O inputs by performing additional bookkeeping and checking operations on each instruction. DIFT has been shown to be quite effective in detecting a large class of common software attacks [1], [2], [3]. Similarly, run-time monitoring and bookkeeping can enable many types of new capabilities such as fine-grained memory protection [4], array bound checking [5], [6], software debugging support [7], managed language support such as garbage collection [8], hardware error detection [9], etc.

This paper presents an architectural framework, named FlexCore, that is designed to enable a large class of run-time monitoring and bookkeeping techniques to execute efficiently in hardware in parallel to the main core. Even though a custom hardware implementation of a particular runtime monitoring scheme would provide the highest efficiency, high development costs and inflexibility of hardware make custom hardware impractical for most run-time monitoring techniques, and especially for techniques developed after the hardware has been fabricated. Modern microprocessor development may take several years and hundreds of engineers from an initial design to production. To justify the costs of development and silicon resources, processor vendors tend to implement a mechanism in custom hardware only if it is already widely used. Moreover, custom hardware mechanisms may become obsolete quickly due to the emergence of new monitoring techniques or the finding of post-production bugs.

Unfortunately, existing proposals for flexible run-time monitoring suffer from either high overheads or limited programmability. For example, one popular approach is to leverage the programmability of an existing processing core and implement a monitoring function in software by instrumenting each dynamic instruction with several more instructions for bookkeeping and checking. Software monitoring approaches offer very high flexibility in allowing monitoring algorithms to be finely tuned long after the chip has been fabricated, however, this flexibility often comes at the cost of drastically reduced application performance. In addition to instruction overheads, a traditional processing core is inherently a poor match for many run-time monitoring schemes that need to perform bit-level operations for bookkeeping. For instance, DIFT needs to propagate and check 1-bit tags on each instruction while a processing core is optimized for sequential 32-bit (or 64-bit) operations. A software implementation for DIFT monitoring on a single core is reported to have an average slowdown of 3.6 times even with aggressive optimizations [10]. Use of multiple cores with specialized hardware support [11] is shown to lower the performance overheads but still requires the use of two large processing cores with specialized logic that will consume more than twice the power. On the other hand, specialized hardware modules [12], [13] are efficient but only support a small subset of monitoring techniques.

In the FlexCore architecture, we propose to combine a bit-level reconfigurable fabric such as an FPGA with

a processing core in order to provide a highly flexible yet efficient parallel monitoring platform. We believe that the FPGA-like fabric is much better suited for run-time monitoring compared to traditional processing cores because many monitoring techniques for security, reliability, and programmability perform bit-level operations in parallel to the main computation. The FPGA-like fabric is also quite general and capable of implementing any computation that fits within the fabric. The FlexCore architecture is a significant deviation from traditional FPGA co-processing, which has been studied as a way to accelerate computations on the main processing core with an explicit control from an application. The FlexCore architecture's reconfigurable fabric is decoupled from the main processing core and supports operations that are parallel and transparent to computation on the main processing core.

In order to minimize the overheads while maintaining the flexibility to support a broad range of monitoring techniques, the FlexCore architecture is carefully designed to match common characteristics of run-time monitoring and hide the inefficiency of reconfigurable fabric. For example, the interface between the processing core and the reconfigurable fabric transparently buffers and forwards dynamic instruction traces relevant to the monitoring technique running on the reconfigurable fabric. The core-to-fabric interface also hides the latencies of cross-clock-domain communications and monitoring operations by decoupling the fabric from the main core through FIFOs.

The FlexCore architecture removes unnecessary overheads for common operations by incorporating custom hardware modules. For example, our architecture includes a dedicated L1 cache for meta-data accesses and a simple filter to selectively forward a subset of instructions. The operations on the reconfigurable fabric are also optimized by incorporating a small shadow register file in the fabric and perform instruction decoding in custom hardware. These architecture features enable efficient implementations of memory arrays and complex decoders, which are particularly inefficient in traditional LUT-based FPGA fabric.

To evaluate the proposed architecture, we implemented the FlexCore architecture and four extensions in RTL (VHDL) based on a simple in-order microprocessor (Leon3), and studied the area, power consumption, and performance on the 65nm process technology. The prototype extensions include uninitialized read checking, DIFT, array-bound checking, and soft-error detection. Evaluation results of the prototype demonstrate that the FlexCore architecture can indeed support a range of extensions with different requirements and operations. Also, the synthesis study shows that the reconfigurable fabric and its interfaces have only minimal impact on the main computing core in terms of its operating frequency and normal operations. In terms of the silicon area, the FlexCore adds about $0.3mm^2$ for dedicated hardware components and all evaluated extensions can fit in a $0.4mm^2$ FPGA fabric, which represent small area overheads for modern processors that are tens of mm^2.

The experimental results also suggest that run-time monitoring on FlexCore is far more efficient than software implementations in terms of both performance and power consumption, and can almost match the performance of ASIC implementations. For example, array-bound checks can be performed by the reconfigurable fabric with a 18% average slowdown and 22% additional power consumption while an ASIC implementation results in a 8% slowdown with 8% additional power consumption. Dynamic Information Flow Tracking (DIFT) on the FlexCore design incurs a 17% slowdown with a 21% power overhead compared to the 5% and 6% overheads of an ASIC. These results are particularly promising because we believe that the prototype implementations can be further optimized. Overall, the prototype implementation shows that the FlexCore architecture is a viable platform at least for in-order processors to provide an efficient and flexible solution for instruction-grained monitoring and bookkeeping.

This paper makes the following main contributions:

- *Computation model*: The paper develops a general co-processing model for instruction-grained monitoring and bookkeeping extensions, and studies how a set of extensions maps to the model. This model forms the basis of the architecture design.
- *FlexCore architecture*: The paper presents the FPGA co-processing architecture that can efficiently implement a broad range of run-time monitoring schemes.
- *Prototype implementations and evaluations*: The paper implements and studies a prototype in RTL (VHDL) and presents results from it.

The rest of the paper is organized as follows. Section II describes the co-processing model for the parallel monitoring and bookkeeping and shows how example extensions can map to the model. Section III presents the FlexCore architecture, and Section IV describes the details of our prototype implementations for both FlexCore and example extensions. Section V studies the performance, area, and power consumption of our prototype extensions in both FlexCore and full ASIC implementations. Section VI discusses the related work, and Section VII concludes the paper.

II. Instruction-Grained Run-Time Monitoring

This section presents the computation model for the fine-grained run-time monitoring and bookkeeping that the FlexCore architecture is designed to support and discusses common characteristics of such co-processing operations. In the following discussion, we use the term "co-processor" to refer to a monitoring and bookkeeping extension.

A. Co-Processing Model

Figure 1 shows the high-level model of how fine-grained monitoring and bookkeeping techniques typically work. In

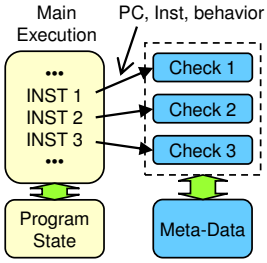


Figure 1. Computation model for parallel monitoring and bookkeeping.

the figure, dark (blue) blocks represent co-processing and light (yellow) blocks represent the main computation. A co-processor maintains its own meta-data, which are often disjoint from the program state, to keep track of the history of computation by the main core. At run-time, the co-processor monitors the execution of the main core at an instruction granularity, updates its meta-data for bookkeeping, and checks certain properties of the main computation. If a check fails, the co-processor may raise an exception. Conceptually, the co-processor observes a trace of all or a subset of instructions and performs its operations on each forwarded instruction. The main core can also communicate with the co-processor with explicit instructions to either configure the co-processor or read information from it.

In general, the run-time monitoring and bookkeeping extensions can be characterized by its meta-data, transparent operations, and software visible operations. Here, we briefly summarize the common characteristics and their implications for the architecture design.

- *Meta-data*: The extensions often need meta-data for both registers and memory. Therefore, the co-processor need to support a memory subsystem for meta-data.
- *Operations*: The monitoring or bookkeeping is often *fine-grained*. For many applications, run-time checks and meta-data updates happen quite frequently, possibly for every instruction on the main core. Therefore, pure software implementations often incur a significant slowdown. At the same time, the operations are largely *decoupled* from the main computation and can be performed in parallel to the main core. The main computation is typically independent from the monitoring extension unless there is an exception. Finally, the extension usually performs *bit operations* rather than 32-bit word operations. Therefore, the operations are a poor match for a regular computing core.
- *Software interfaces*: The co-processor needs to be able to raise an exception and communicate with the main core with explicit instructions from the core. The instructions may read/write configuration registers on the co-processor and/or perform custom operations for each extension. The exception and the explicit instructions are usually infrequent and do not have to be fast.

B. Example Monitoring Extensions

This subsection presents a set of monitoring and bookkeeping extensions as examples and show how those extensions map to our co-processing model. Table I summarizes the operations of four example extensions: UMC, DIFT, BC, and SEC. Section IV describes more details of each extension and its implementation.

Uninitialized Memory Check (UMC): Uninitialized Memory Check (UMC) is a simple mechanism that is widely used for software debugging to ensure that a memory location is initialized (written) before a read. For each memory location, the UMC mechanism maintains a one-bit tag that represents whether the location is initialized or not. The tag is set after a write to the corresponding memory location and cleared with an instruction from the main core when the corresponding memory is de-allocated. On a load operation, the mechanism checks the tag and raises an exception if the memory location is not initialized.

Dynamic Information Flow Tracking (DIFT): Dynamic Information Flow Tracking (DIFT) detects software attacks by tracking potentially malicious values from I/O and checks their uses. DIFT can detect low-level exploits such as buffer overflows and format string attacks [1], [16], [2], without any modifications to executables, and detect high-level attacks such as SQL injections and cross-site scripting with simple application-level checks [3]. For DIFT, a co-processor needs meta-data (called taint tags) to indicate the source of each value in registers and memory, and explicit instructions so that the OS on the main core can set or clear those taint tags. For ALU, load, and store operations, the co-processor propagates taint tags from the source operand(s) to the destination; an OR operation of the source tag bits determine the destination tag. On security critical operations such as indirect jumps, the co-processor checks the tag and raises an exception if tainted values are used.

Array Bound Check (BC): An array bound check (BC) is a popular way to detect spatial memory errors such as buffer overflows, which account for a large portion of software errors and vulnerabilities in unsafe languages such as C. For BC, a co-processor keeps meta-data that encode the array bounds for a pointer and checks if each memory access is within the bounds. The array bounds can be expressed in various ways including an object table [17], [18], base and bound addresses [19], [5], or color tags [6], [20]. In our prototype, we encode bounds by assigning a color tag to each pointer and memory location. On a memory allocation, a program marks the resulting pointer and the corresponding memory locations with an identical tag using special instructions. For each memory access, the co-processor checks whether the pointer tag matches the memory location tag, and raises an exception if not.

Soft Error Check (SEC): Recently there have been significant efforts to develop architectural techniques to detect hardware errors such as a transient bit-flip. In a high-

Extension	Meta-Data	Transparent Operations	SW Visible Operations
UMC [14]	1. 1-bit tag per word in memory.	1. Set the tag on a store. 2. Check the tag on a load.	1. Clear tags on a de-allocation. 2. Exception when a tag check fails.
DIFT [1]	1. 1-bit tag per register. 2. 1-bit tag per word in memory.	1. Propagate tags on ALU/load/store. 2. Check tags on a control transfer.	1. Set tags for values from I/O. 2. Clear tags on a declassification. 3. Set a security policy register. 4. Exception when a tag check fails.
BC [6]	1. 4-bit tag per register. 2. 8-bit tag per word in memory.	1. Propagate tags on ALU/load/store. 2. Check a pointer tag (register) with a memory tag on a load/store.	1. Set reg/mem tags on array allocation. 2. Clear tags on a de-allocation. 3. Exception when a tag check fails.
SEC [15], [9]		1. Check an ALU operation.	1. Exception when a check fails.

Table I
EXAMPLE FLEXCORE CO-PROCESSING EXTENSIONS. UMC: UNINITIALIZED MEMORY CHECK. DIFT: DYNAMIC INFORMATION FLOW TRACKING, BC: ARRAY BOUND CHECKING, SEC: SOFT ERROR CHECKING.

level, these techniques either re-execute each instruction in parallel [15] or check simpler checksums on each operation [9]. These soft-error checks can be easily mapped to the FlexCore model. As an example, consider a simple checker that verifies the result of each ALU operation by computing checksums as proposed by Argus [9]. A co-processor performs the checksum operation on each ALU instruction using the source and result values from the main core, and raises an exception if the check fails.

Other Extensions: We believe that the FlexCore co-processing model will be applicable to a large class of hardware extensions that perform monitoring and/or bookkeeping operations in parallel to the main computation. As an example, the co-processing model can support simple profiling applications such as custom performance monitors and detailed analysis of software characteristics. The co-processing model also supports various techniques to enhance software security and reliability, including fine-grained memory protection [4], debugging support [7], checkpointing [21], and others. Parallel bookkeeping can also provide an efficient support for high-level language features such as garbage collection [8]. Finally, the reconfigurable co-processor may be used as an accelerator for the main computation as in traditional proposals [22].

III. FLEXCORE ARCHITECTURE DESIGN

This section describes the FlexCore architecture. The description focuses on how the FlexCore architecture is designed to enable run-time monitoring and bookkeeping techniques in an efficient manner exploiting the common characteristics discussed in the previous section.

A. Scope and Design Goals

This paper illustrates the overall architecture in the context of a single-issue processor without multi-threading. We note that recent studies on run-time monitoring on multi-cores also focus on a simple core as the first step [11]. While the general architecture is also applicable to superscalar processors, further performance optimizations may be necessary for high-performance processors. Multi-threading introduces a coherence issue between program data and meta-data. This paper does not discuss this issue in detail because it is common for many run-time monitoring techniques with meta-data and has been studied already [13], [23].

To be useful in practice, the FlexCore architecture needs to be both flexible and efficient, and have minimal impacts on the main processing core. The following list summarizes our main design goals:

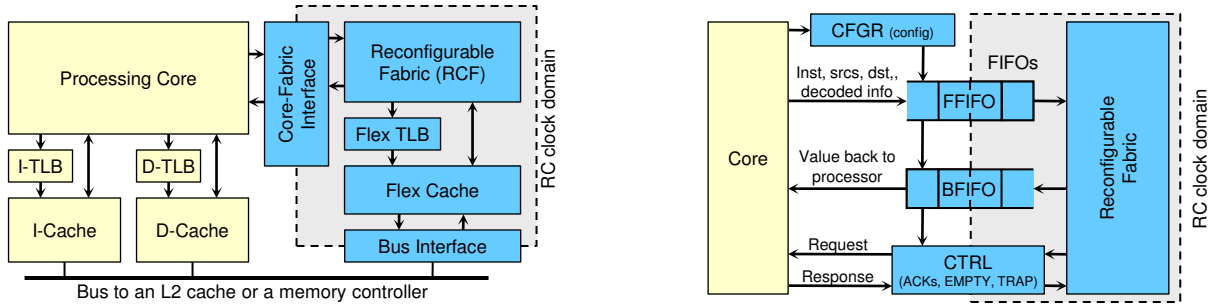
- *Flexible*: The FlexCore architecture should support a broad range of run-time monitoring and bookkeeping extensions so that even new techniques can be implemented in the future. The architecture should also allow extensions to be updated or added in the field.
- *Efficient*: Extensions on FlexCore should be much more efficient than software implementations especially in terms of performance and energy consumption.
- *Non-Intrusive*: The added programmability should not degrade the performance of a processor when the reconfigurable fabric is not in use. To minimize the implementation costs, the architecture should only require minimal changes to today's microprocessor pipeline.

B. Architecture Overview

Figure 2(a) shows the high-level block diagram of the FlexCore architecture. The yellow (light) rectangles represent components in traditional microprocessors and the blue (dark) rectangles represent new components for FlexCore. In a high-level, the architecture closely resembles the co-processing model that is described in the previous section. The main processing core forwards its execution trace to the reconfigurable fabric for parallel monitoring and bookkeeping and receives signals from the fabric through the interface module. The architecture provides a separate memory subsystem for meta-data with its own L1 cache and optionally a TLB if virtual memory is supported. The meta-data shares a lower level memory hierarchy such as an L2 cache and the main memory with program data.

The architecture is carefully designed to exploit the common characteristics of run-time monitoring techniques without restricting the specifics of the monitoring operations. For example, unlike traditional FPGA co-processors, the main core forwards its execution trace without explicit intervention from software. The architecture is also optimized for *bit operations* that are common in run-time monitoring. The reconfigurable fabric operates at a bit granularity and the meta-data cache supports bit-level writes.

While the fine-grained reconfigurable fabric is a good fit for typical meta-data computations in run-time monitoring



(a) Overview (b) Core-fabric interface

Figure 2. FlexCore architecture block diagrams.

schemes, the bit-level reconfigurability is a poor match for coarse-grained structures such as memory arrays. The FlexCore architecture handles inefficiencies in fine-grained reconfigurable fabric with a set of optimizations.

- *Specialized hardware modules*: The architecture incorporates a set of dedicated hardware units for functions that are common across many extensions. These modules include the core-fabric interface, TLB, cache, and meta-data register file.
- *Filtering and pre-processing*: The core-fabric interface reduces the workload of the reconfigurable fabric by filtering out unnecessary types of instructions and performing decoding of instructions.
- *Decoupled executions*: The reconfigurable fabric is decoupled from the main core through FIFOs. The processing core forwards its execution trace, but does not need to wait for an acknowledgment from the fabric in most cases. This decoupling hides latencies from co-processing as well as communications across clock domains. The clock domain of the reconfigurable fabric is carefully selected to include the TLB and the cache so that TLB and cache hits do not require cross-domain communications.

C. Core-Fabric Interface

The reconfigurable fabric communicates with the main core through a set of FIFO interfaces as shown in Figure 2(b). The FIFOs are connected to/from the commit stage of the main core. The details of the core-to-fabric interface are also listed in Table II. The core-to-fabric interface works to enable fine-grained instruction communication between the core and reconfigurable fabric. The processing core sends its execution trace to the co-processor using the FIFO interface, so that the fabric can perform monitoring or bookkeeping operations on each forwarded instruction.

A forward FIFO sends a trace of instructions, which are completed and ready to commit, in the program order. A FIFO packet contains fairly comprehensive information, including a program counter, source and destination register values, ALU results, condition codes, and branch outcome. The FIFO packet also includes decoded instruction fields such as an opcode, source register numbers, and a destination register number, in order to reduce the burden on the reconfigurable fabric. For comparison, we found that our DIFT prototype can run 30% faster by performing the instruction decoding for operands and control signals on the core side.

A forwarding configuration register (CFGR) specifies how a forward FIFO handles each instruction type[1]. For example, the CFGR can be configured to forward load/store instructions but ignore ALU or control instructions when implementing uninitialized memory checking. The architecture provides three choices regarding whether an instruction should be forwarded or not. A FIFO can be configured to (i) ignore an instruction, (ii) forward an instruction only if a FIFO entry is available (ignore if the FIFO is full), or (iii) always forward an instruction. The third choice implies that an instruction commit is stalled if the FIFO is full. The FIFO may also be configured to either allow an instruction to commit as soon as it is enqueued or stall the commit until there is an acknowledgment from the co-processor.

In many co-processor extensions, the main core does not need to wait for the reconfigurable fabric because an exception from the co-processor does not need to be precise. For example, all four extensions in our prototype (UMC, DIFT, BC, and SEC) terminate a program if a check fails. There is no need to support a restart on these extensions. If an instruction requires a value from the fabric as in the "read from co-processor" instruction or if an exception from the fabric must be precise, the main core needs to delay a commit operation. For modern out-of-order processors, such delays simply mean that instructions stay in an ROB (Re-Order Buffer) longer without necessarily stalling the following instructions. In-order cores can either stall an instruction at the commit stage or add a simple roll-back mechanism to provide a precise exception and allow instructions to speculatively commit.

The reconfigurable fabric uses additional FIFOs to communicate back to the main core. A back FIFO (BFIFO) sends a return value for the "read from co-processor" instruction. In addition to data, the control module (CTRL) allows a set of synchronization operations between main core and the co-processor. The co-processor sends an acknowledgment

[1]There are 32 types in our prototype based on the SPARC architecture.

Function	Module	Field	Description	Bits
Config	CFGR	FFIFO	Select a FIFO behavior for each instruction type: 1) ignore, 2) accept only if not full, 3) accept and proceed, 4) accept and wait for an acknowledgement. Contains 2 bits for each of the main 32 instruction types (SPARC prototype).	64
Core To Fabric	CTRL	PACK	Acknowledgement for a trap signal from the co-processor.	1
	FFIFO	PC	Program counter.	32
		INST	Undecoded instruction.	32
		ADDR	Address for a load/store.	32
		RES	Result of an instruction.	32
		SRCV1	Source operand 1 value.	32
		SRCV2	Source operand 2 value.	32
		COND	Condition codes that affect instruction processing.	4
		BRANCH	Computed branch direction information.	1
		OPCODE	Decoded instruction opcode.	5
		DECODE	Miscellaneous decoded signals.	32
		EXTRA	Extra processor control signals.	32
		SRC1	Decoded Source1 register number.	9
		SRC2	Decoded Source2 register number.	9
		DEST	Decoded Destination register number.	9
Fabric To Core	CTRL	CACK	Acknowledgement for FFIFO.	1
		EMPTY	A signal to indicate that there is no pending instruction in the co-processor.	1
		TRAP	Raise en exception.	1
	BFIFO	VAL	A return value on a 'read from co-processor' instruction.	32

Table II
THE FLEXCORE INTERFACE BETWEEN THE CORE AND THE FABRIC.

back (CACK) for an instruction when the commit stage in the main core waits for a completion of the Flex fabric processing. On an exception or a trap on the main core, the core needs to wait for the co-processor to finish all pending instructions before starting the handler. For this purpose, the fabric provides a signal (EMPTY) to indicate whether there are any pending instructions in the co-processor. The reconfigurable fabric can also raise an exception using the trap signal (TRAP). If the interrupt level of the processor is sufficiently low, the main core acknowledges such an exception (PACK) and invokes a proper handler.

Note that the proposed FIFO interface with the reconfigurable fabric can easily support custom instructions on the main core for each extension. For example, in order to implement an instruction to set a configuration register within the reconfigurable fabric, the fabric can be programmed to update the register on a particular instruction encoding.

D. Meta-Data Memory Hierarchy

For meta-data used by the reconfigurable fabric for bookkeeping, the reconfigurable fabric uses its own cache subsystem that is separate from the main core's L1 caches. This design minimizes changes to the main core's cache structures. Both the processing core and the reconfigurable fabric share the lower-level memory hierarchy such as an L2 cache and main memory. Currently, the architecture does not maintain coherency between the main core's L1 caches and the meta-data L1 cache. For the extensions that we studied, the co-processor only need to access meta-data in memory regions disjoint from program instructions and data. The architecture can be extended with a cache coherence mechanism if necessary.

The meta-data cache is almost identical to regular data caches except for the capability to write at a bit granularity. Meta-data cache reads return 32-bit words as in regular caches. For writes, the meta-data cache is given a 32-bit write enable mask in addition to an address and a data word, and only updates bits within the cache word where the bit mask is set. We found that the bit-level write capability is essential for efficient co-processing since many co-processing techniques work on meta-data much smaller than a word. Without this feature, a co-processor needs to perform an explicit cache read and then an explicit cache write in order to update meta-data.

E. Reconfigurable Fabric Architecture

The high-level FlexCore architecture is independent from the micro-architecture of the reconfigurable fabric and is applicable to various types of fabrics. In this paper, we use a standard LUT-based FPGA architecture, specifically the Xilinx Virtex-5, which includes standard Configurable Logic Blocks (CLBs) with LUTs and flip-flops. The FPGA fabric is chosen over other coarse-grained fabrics because typical monitoring extensions perform bit-level operations. Our reconfigurable fabric also includes an embedded meta-data register file, which is implemented with custom hardware and has an 8-bit shadow register for each general-purpose architecture register in the main core. The register file provides an efficient way to keep meta-data for registers in a similar way that SRAM banks in commercial FPGAs provide efficient memory blocks.

F. Programming Reconfigurable Fabric

The FlexCore architecture is designed for scenarios where the co-processing extension is treated as a hardware extension that only need to change infrequently. For example, we envision that the reconfigurable fabric is programmed at the boot time and not reprogrammed until the next time power is cycled. In this context, the programming of the fabric does not need to be fast, and we can use the standard programming methods in today's commercial FPGAs where a bitstream is serially shifted in to configuration memory.

The reconfigurable fabric can closely monitor virtually all computations of the main core, therefore, its programming must be restricted to only trusted parties for security and privacy. A processor vendor can choose from two options depending on how open the FlexCore feature is desired to be. First, the FlexCore programming can be treated similar to today's microcode updates. Only the vendor can create a valid update for the reconfigurable fabric and the programming interface is invisible to the software layer including an operating system. In this way, a vendor can tightly control FlexCore extensions. Alternatively, the FlexCore programming interface can be exposed to an operating system. In both cases, the operating system must properly manage the meta-data memory space and ensure memory isolation between the meta-data of each process.

IV. Prototype Implementations

To evaluate the FlexCore architecture, we built a prototype system based on the Leon3 microprocessor [24]. Leon3 is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The Leon3 architecture provides a single-issue in-order pipeline with seven stages. The core-fabric FIFO interfaces are added to the exception stage of the pipeline. In the prototype, there is no L2 cache as the Leon3 processor does not include one. As this paper focuses on the effectiveness of the high-level FlexCore architecture, the current prototype does not support the meta-data TLB for multi-programming.

The rest of the section describes the details of the four monitoring and bookkeeping extensions that we presented in Section II. The extensions range from simple ones such as UMC to more sophisticated and area intensive ones such as BC. While not discussed in detail due to the space constraint, we note that extensions are moderately pipelined (3 to 6 stages) to improve the throughput.

A. Uninitialized Memory Check (UMC)

UMC uses a 1-bit meta-data tag for each memory word to check if a word has been initialized. The tag is updated on a store instruction and checked on a load instruction. Software explicitly clears tags on memory de-allocation. Figure 3(a) shows the block diagram of the UMC extension. For UMC, the core-to-fabric interface gets configured to forward load or store instructions along with special communication instructions. The rest is ignored. From each FIFO entry, the extension takes an opcode and a memory address. The address gets translated to the address of the 1-bit tag by shifting and adding to a base, and is used to access the meta-data cache. On a store, the tag is updated to 1. On a load, the tag from the cache is checked. If the tag is not set, an exception is raised through the control module.

B. Dynamic Information Flow Tracking (DIFT)

In FlexCore, DIFT can be implemented by having a co-processor maintain taint bits and perform the taint propagation and checks. For meta-data, the co-processor needs to keep a taint tag for each architecture register and word in memory. For simplicity, our prototype uses a 1-bit tag per word, which is enough to detect attacks[2]. The co-processor needs to perform taint propagation and check operations on each instruction from the main computation. For ALU, load, and store operations, the taint tags get propagated from the source operand(s) to the destination; an OR operation of the source tag bits determine the destination tag. On security critical operations such as indirect jumps, the co-processor checks the tag and raises an exception if tainted values are used. To interface with software, the DIFT co-processor needs to support a few new instructions to explicitly set/clear taint tags and set control registers, which specify the tag propagation and check policies.

The block diagram for DIFT is shown in Figure 3(b). Because our DIFT prototype maintains a 1-bit tag for each word, the internal data-paths are 1-bit in width. DIFT keeps a register file with 1-bit entries and uses the cache to access the 1-bit meta-data in memory. In DIFT, the core-to-fabric FIFO is configured to forward loads, stores, ALU instructions, indirect jumps, and co-processor instructions. For loads and stores, the DIFT extension uses the opcode, the memory address, and decoded register numbers from the FIFO to move a tag from memory to register or from register to memory. The memory address gets translated in the same way as UMC. For ALU operations, the extension uses the opcode and the register numbers to read tags from the internal registers and update the destination register. The tag is checked on indirect jumps. The DIFT extension allows software to set its control registers and tags through explicit instructions.

C. Array Bound Check (BC)

In this study, we use a bound-checking technique based on colors [6] as an example extension. The technique detects an out-of-bound accesses by assigning a color tag to each pointer and memory location. On a memory allocation such as `malloc()` for the heap and a function prologue for the stack, a program marks the resulting pointer and the corresponding memory locations with an identical color using special instructions. During the execution, the pointer carries the color. For each memory access, the pointer color is matched to the memory location color to ensure an in-bound access. This approach can detect spatial memory errors by assigning different colors to different objects.

The color-based bound checking can be implemented in the FlexCore co-processing model in a way similar to UMC and DIFT. In fact, the bound checking can be seen as a combination of UMC and DIFT features. For meta-data, the bound checking co-processor needs to maintain one tag for each register and word in memory, which represents

[2]DIFT implementations may use multiple bits per tag, or have a tag per each byte in memory. However, the basic operations are identical and this discussion applies to those variants in the same way.

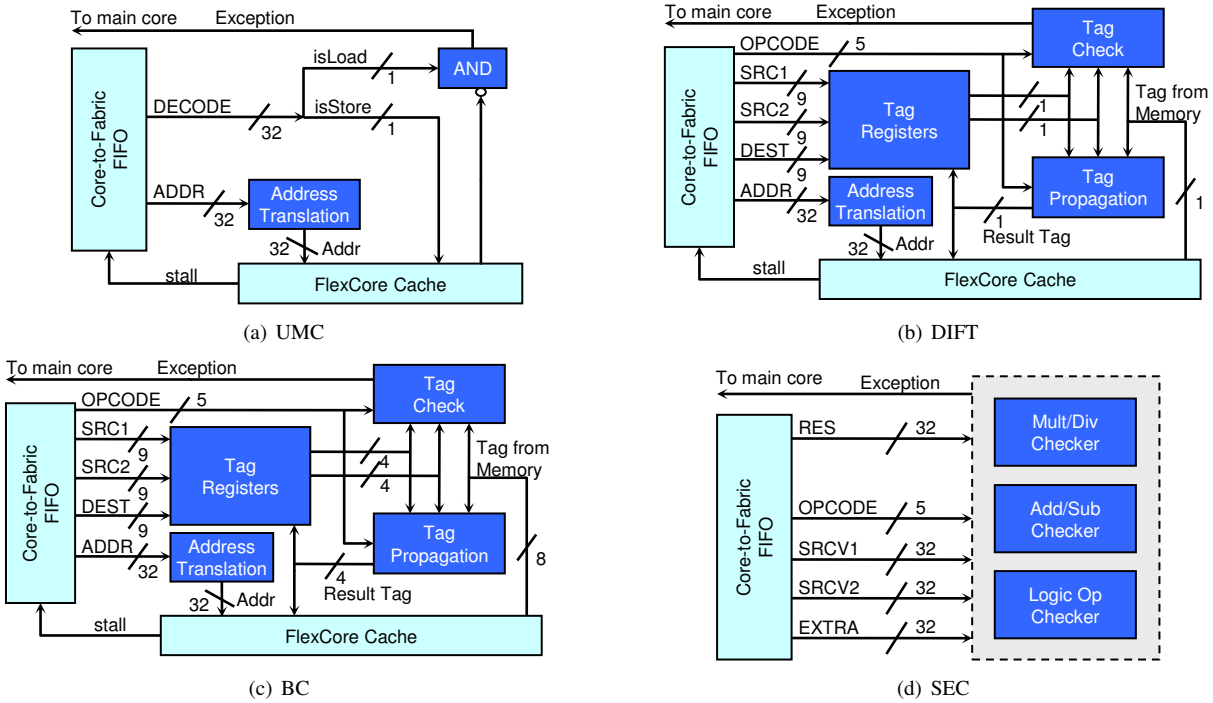


Figure 3. Block diagrams for FlexCore extension prototypes. Dark blocks represent the FPGA fabric.

the color of a pointer value (pointer tag). For each word in memory, the co-processor keeps an additional tag to represent the location color (memory tag). The tags are set by the main computation using co-processor instructions. The co-processor propagates the pointer tag on an ALU, load, or store operation by either copying the tag (load/store) or adding two source tags (ALU ops). On a memory operation, the co-processor checks if the pointer tag matches the memory tag, and raises an exception for an out-of-bound access if they do not match.

The high-level block diagram of BC is virtually identical to DIFT as shown in Figure 3(c). Internally, however, BC keeps two 4-bit meta-data tags for each word, one for the pointer color and the other for the memory color. The BC extension maintains a register file with 4-bit entries and 8-bit tags for each word in memory. Conceptually, operations for the pointer tags are similar to DIFT and operations for the memory tags are similar to UMC. In BC, the core-to-fabric FIFO is configured to send loads, stores, arithmetic instructions (for pointers), and co-processor instructions.

For a load instruction from the main core, the BC extension loads an 8-bit tag from the meta-data cache after the address translation. The lower 4 bits represent a memory tag and are compared with the tag of the address to detect an out-of-bound read. The upper 4 bits represent a pointer tag and are copied into the tag register for the destination. Similarly, on each store instruction, the BC extension checks if the access matches the memory tag, and copies the pointer tags of the source register to the memory. For arithmetic instructions, the extension propagates the pointer tags from the source registers to the destination register. The BC extension raises an exception if the tags do not match on a load or store instruction.

D. Soft Error Check (SEC)

For hardware error detection, we implemented a checker that verifies the computation result from the main core's ALU. While verifying other aspects of the execution such as instruction decoding will require more logic, we believe that the ALU checker represents the general characteristics of soft error checks where verification of each instruction is independent. Figure 3(d) shows the block diagram of the SEC extension. For SEC, the core-to-fabric FIFO sends all ALU instructions along with their opcodes, source operand values, and results. The reconfigurable fabric checks the result from the main core by effectively re-executing the computation as in Argus [9]. The checker verifies each bit individually for addition/subtractions and logic operations. For efficiency, we used modular arithmetic (mod M) to verify multiplications and divisions, where M is a Mersenne number of 3. For the computation checks, the extension mainly uses the opcode, the source values, and the result value from the core-to-fabric FIFO. If the checking engine detects an error, the SEC extension raises an exception.

V. Evaluation

This section evaluates the proposed FlexCore architecture using silicon area, power consumption, and performance as the metrics. To study the overheads of the reconfigurability, FlexCore is compared to ASIC implementations of each extension. We do not evaluate the functional effectiveness

of each extension, such as the attack detection capability of DIFT, as that has already been demonstrated previously.

A. Methodology

To estimate the area, power consumption, and operating frequency of the hardware modules for a typical ASIC flow, we used Synopsys Design Compiler (DC) with a 65nm IBM technology library. To estimate the same metrics for extensions implemented on the FPGA fabric, we used Synplify Pro and Xilinx ISE. The tools were used to map each extension to the Virtex-5 FPGA, which was also manufactured in a 65nm technology. Virtex-5 was chosen to match the technology node that we use in the ASIC flow. Each extension on an FPGA was fully placed and routed, without dedicated components such as the core-fabric interfaces and caches. This synthesis provided the estimates for the operating frequency and the number of LUTs. To compute a rough estimate of the area, we used an estimate of CLB tile area from the model by Kuon and Rose [25]. The model reports that the area of a CLB tile with 10 6-input LUTs in the 65nm technology node is approximately 8,069μm^2. We used this estimate of 807μm^2 per LUT and multiplied it by the total number of LUTs used in our design to generate an area estimate. To compute an estimate of the power of the reconfigurable fabric, we used the Virtex-5 Power Spreadsheet [26] using the frequency and number of LUTs from FPGA synthesis. The area and power consumption of the shadow register file are obtained from a memory compiler and included in the estimates for the dedicated FlexCore modules. The power estimates currently use a fixed toggle rate of 0.1 and static probability of 0.5 for both ASIC and FPGA to provide rough comparisons.

To evaluate the impact of the reconfigurable extensions on the overall performance, we performed RTL simulations of the entire system including the processing core, caches, a FlexCore extension, a memory controller, and off-chip SDRAM. The default configuration includes a Leon3 core with a single-issue 7-stage pipeline, 32-KB L1 instruction and data caches with 32-B lines, a 4-KB meta-data cache with 32-B lines, and a 64-entry core-to-fabric FIFO. The Leon3 caches use a write-through with no allocate policy. The simulations ran a set of benchmark programs from MiBench [27] and small kernels.

B. Area, Power, and Frequency

Table III summarizes the estimated area, power consumption, and operating frequency for the Leon3 processor with and without various extensions. We found that the unmodified Leon3 with 32-KB L1 caches can run up to 465MHz and consume about 0.836mm^2 and 364.2mW. The full ASIC results, where the Leon3 processor with each extension is synthesized using the ASIC flow, show that UMC, DIFT, and BC consume 12 to 20% additional silicon area and 6 to 8% additional power. These overheads are dominated by the meta-data cache and FIFOs for the core interface. For SEC, the overheads are negligible because SEC does not require a meta-data cache or a complex interface. The Leon3 processor with an extension in full ASIC implementations results in a slightly lower operating frequency because the extensions tap into internal pipeline signals.

For the FlexCore implementations, the table separately shows the estimates for each extension on the Flex fabric and the estimates for the dedicated (ASIC) modules common for all FlexCore implementations including the FlexCore interface and 4-KB meta-data cache. Similar to the ASIC implementations, the synthesis results show that the addition of the FlexCore interface that taps into the main core pipeline slows down the frequency slightly by 1.5%. The dedicated FlexCore modules (the interface and the meta-data cache) add about 32.5% more silicon area and 14.6% more dynamic power compared to the baseline Leon3 processor. These overheads are higher than the ASIC implementations because the FlexCore interface is more general. In the FlexCore implementations, the Flex fabric adds noticeable overheads in addition to the meta-data cache and the interface. The extensions require the Flex fabric to be 0.09 to 0.39mm^2, which represent 11 to 47% additional area overheads, and consume 6 to 10% additional power.

While the relative area overheads are noticeable for the Leon3 processor, which is a tiny embedded processor, the results demonstrate that the FlexCore architecture is far more energy-efficient than running a monitoring function on another processing core. Also, we note that the absolute area and the power consumption for the FlexCore modules are quite small if compared to higher-end microprocessors. For example, each processing core in UltraSPARC T2 occupies 12mm^2 in the 65nm technology. MIPS R14000, which is fabricated in the 0.13μ technology, occupies 204mm^2 and consumes 17W at 500MHz. For these modern processors, the relative overheads of FlexCore will be insignificant.

C. Performance

Table IV presents the normalized execution time for each extension. The execution time is normalized to the baseline Leon3 without any extension. The ASIC implementations with the same clock frequency for both an extension and the core show at most 7% performance overheads. The overheads come from two sources, stalls from a full forward FIFO and contention for the shared memory bus. A cache miss on a meta-data access forces the extension to stall while meta-data is refilled from memory. During this time the forward FIFO fills up with waiting instructions and may stall the main core if it becomes full. Also, meta-data refills from memory hog the memory bus shared by the meta-data cache and the main core caches. Therefore, the main core suffers from increased memory access latencies for its own cache misses.

	Extension	Description	Max Freq (MHz)	Area		Power	
				μm^2	overhead	mW	overhead
Baseline	-	Unmodified Leon3 w/ 32KB L1	465	835,525	-	365	-
ASIC	UMC	Leon3 w/ UMC	463	932,118	11.6%	388	6.3%
	DIFT	Leon3 w/ DIFT	456	960,558	15%	388	6.3%
	BC	Leon3 w/ BC	456	996,894	19.3%	393	7.7%
	SEC	Leon3 w/ SEC	463	836,786	0.15%	364	-
FlexCore	Common	Leon3 w/ dedicated FlexCore modules	458	1,106,967	32.5%	418	14.6%
	UMC	UMC on Flex fabric (FPGA)	266	90,384	10.8%	21	5.8%
	DIFT	DIFT on Flex fabric (FPGA)	256	123,471	14.8%	23	6.3%
	BC	BC on Flex fabric (FPGA)	229	203,364	24.3%	27	7.4%
	SEC	SEC on Flex fabric (FPGA)	213	390,588	46.7%	36	9.9%

Table III
THE AREA, POWER, AND FREQUENCY OF THE FLEXCORE ARCHITECTURE. THE OVERHEADS IN SILICON AREA AND POWER CONSUMPTION ARE SHOWN RELATIVE TO THE BASELINE LEON3.

Benchmark	UMC			DIFT			BC			SEC		
	(1X)	(0.5X)	(0.25X)	(1X)	(0.5X)	(0.25X)	(1X)	(0.5X)	(0.25X)	(1X)	(0.5X)	(0.25X)
sha	1.01	1.01	1.01	1.01	1.06	1.16	1.03	1.07	1.15	1.00	1.33	1.50
gmac	1.01	1.01	1.09	1.01	1.15	1.34	1.02	1.17	1.37	1.00	1.20	1.47
stringsearch	1.03	1.05	1.12	1.16	1.46	1.89	1.22	1.45	1.84	1.00	1.00	1.11
fft	1.01	1.01	1.01	1.02	1.05	1.31	1.02	1.03	1.35	1.00	1.15	1.45
basicmath	1.01	1.01	1.01	1.03	1.08	1.34	1.04	1.07	1.37	1.00	1.14	1.43
bitcount	1.04	1.06	1.07	1.08	1.36	1.69	1.13	1.27	1.64	1.00	1.19	1.48
geomean	1.02	1.02	1.05	1.05	1.18	1.43	1.07	1.17	1.44	1.00	1.16	1.40

Table IV
THE PERFORMANCE OVERHEAD COMPARISONS BETWEEN ASICS AND FLEXCORE. THE PERFORMANCE IS SHOWN AS THE EXECUTION TIME THAT IS NORMALIZED TO THE EXECUTION TIME OF THE BASELINE LEON3 PROCESSOR WITHOUT MODIFICATIONS. THE FLEX FABRIC RUNS AT HALF THE CLOCK FREQUENCY AS THE MAIN CORE FOR DIFT, UMC, AND BC, AND RUNS AT ONE QUARTER OF THE MAIN CORE FREQUENCY FOR SEC.

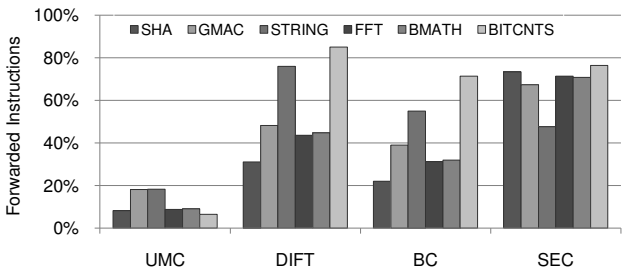


Figure 4. The percentage of instructions forwarded to the reconfigurable fabric for each FlexCore extension prototype.

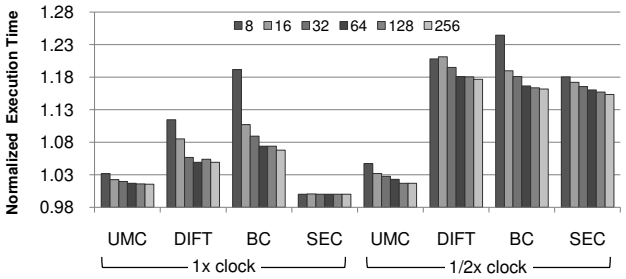


Figure 5. Average FlexCore performance for various forward FIFO sizes.

The performance overheads of the FlexCore implementations are estimated by running RTL simulations with two separate clocks for the main core and the Flex fabric. The clock frequency for the Flex fabric is set based on the frequency estimates from the synthesis results. BC, UMC, and DIFT run at half the frequency as the main core (0.5X in the table) while SEC runs slower (0.25X). For UMC, the performance of FlexCore is virtually identical to the ASIC performance despite running at half of the core frequency because only a small portion of instructions are forwarded to the reconfigurable fabric to be processed as shown in Figure 4. BC and DIFT have a slight higher performance overheads of 18% for FlexCore because the fabric needs to process a larger percentage of main core instructions and access the memory for meta-data. However, the FlexCore overheads are still quite low. SEC has the highest performance overheads at 40% because it processes a large number of instructions at a low clock frequency.

The experimental results demonstrate that the FlexCore extensions are far more efficient than software implementations with comparable capability. For DIFT, previous software implementations have placed the performance overhead as high as 37 times [2]. Even a highly optimized implementation on high-end superscalar processors reported an average slowdown of 3.6 times [10]. For UMC, Purify [14] performs similar checks by adding state bits to each byte in memory, and was reported to be up to 5.5 times slower. The array bound check in software can also incur a noticeable slowdown in memory intensive programs up to 1.69 times even with extensive optimizations [18]. We also note that these software implementations are tested on high-performance processors where additional instructions can be hidden by the superscalar and dynamic scheduling techniques. We expect the software overheads to be even higher for simple in-order processors.

Since the processor must stall when a forward FIFO between the core and fabric is full, the FIFO must be sized appropriately to accommodate the slowdowns on the reconfigurable fabric for meta-data accesses. For this purpose, we studied the impact of the FIFO size on the performance for our extensions as shown in Figure 5. A 64-entry forward

FIFO was found to be sufficient for our implementation. FIFO sizes smaller than 64 entries caused noticeably greater performance overheads while larger forward FIFO sizes offered marginal performance benefits. The silicon area for the FIFO only increase by about 10% between the 16-entry FIFO and the 64-entry FIFO because of the SRAM peripheral circuits.

VI. Related Work

This section summarizes the existing FPGA co-processor architectures and other programmable run-time monitoring architectures. To the best of our knowledge, this work is the first that proposes to use the on-chip reconfigurable fabric such as an FPGA as a general framework for run-time monitoring and bookkeeping. The use of bit-level reconfigurable fabric enables FlexCore to support a large class of extension with low energy and performance overheads.

On-chip reconfigurable fabric for co-processing: The integration of a reconfigurable fabric into a microprocessor has been extensively studied in the context of speeding up the main computation. For example, Chimaera [28] and others [29], [30] propose to integrate reconfigurable functional units in the processor pipeline. On the other hand, another set of approaches such as Garp [22] and OneChip [31] propose a reconfigurable unit as an on-chip accelerator. While we also propose an on-chip co-processor, our goal is to enable parallel monitoring functions that are largely decoupled from the main computation and cannot be efficiently supported by the traditional FPGA co-processors.

Run-time monitoring on multi-cores: Recently, researchers have proposed to utilize idle cores on many-core processor for run-time monitoring of security and reliability properties. For example, INDRA [32] uses a checker core to monitor coarse-grained events on a computation core such as function call/return, code origin inspection, and control flow inspection. Nagarajan et al. studied implementing DIFT on multi-cores [33]. Unfortunately, because the checker core needs to run multiple instructions to process each event from the computation core, these early designs are either limited to coarse-grained monitoring or incur significant slowdowns (3 to 10x). To address such inefficiencies, Chen et al. proposed a set of hardware acceleration techniques for run-time monitoring on multi-cores [11]. However, their architecture still resulted in non-trivial slowdowns for fine-grained extensions. Moreover, in these approaches, the checker core consumes the same amount of power as the main core. While the multi-core approaches aim to support similar run-time monitoring schemes, FlexCore provides higher flexibility and efficiency because its fabric can be programmed at a bit-level granularity and perform multiple operations in parallel.

Programmable security and reliability extensions: Rather than being general, some of the previous techniques target to efficiently support a small set of run-time monitoring techniques. MemTracker [12] uses a programmable finite state machine and tags in memory to support extensions that monitor memory accesses. For example, MemTracker can perform the uninitialized memory check and a simple array bound check by placing inaccessible guard words at the end of an array. As another example, FlexiTaint [13] supports DIFT operations with a fully programmable tag propagation and check policies. While these techniques are more efficient than FlexCore on extensions that they are designed for, FlexCore is much more general and supports a larger class of extensions.

3-D introspection: Recently, researchers have started to investigate using the 3-D stacking technology to monitor operations of a microprocessor [34]. In this approach, a new hardware feature is implemented in its own die and gets stacked on top of a microprocessor. In a high-level, both the 3-D introspection and FlexCore aim to realize similar run-time monitoring functions. The main difference lies in where the extensions are implemented: either on a separate die or on the same die with the processor. The Flex architecture is likely to be more efficient for fine-grained checks requiring significant communications whereas the 3-D introspection is likely to be better suited for compute-intensive monitoring that is inefficient for the FPGA-like fabric.

Post-silicon validation and repair: Researchers have proposed to add an on-chip reconfigurable processing engine that monitors internal processor signals for the purpose of facilitating post-silicon debugging [35], [36] or patching hardware design flaws [37]. While these approaches also perform transparent run-time monitoring in a high-level, FlexCore resulted in a significantly different design because of the difference in target applications.

VII. Conclusion

In this paper, we proposed FlexCore, a hybrid architecture which combines a microprocessor with an on-chip reconfigurable fabric. FlexCore is designed to be a general platform where hardware run-time monitoring features can be added post-fabrication. Our case studies and prototypes of four extensions show that FlexCore can support a range of monitoring extensions from simple uninitialized memory checks to more complex information flow and soft error checks. The careful use of dedicated modules along with the fact that these extensions often perform simple bit-level operations enable the FlexCore architecture to efficiently support these monitoring extensions. Our performance evaluation results show that the architecture can almost match the performance of full-custom ASIC implementations.

As future work, we plan to study more extensions, investigate how the FlexCore approach can be applied to high-performance superscalar cores where multiple instructions may execute in parallel, and investigate other reconfigurable fabric architectures to further improve the performance. We also plan to extend the architecture to support multi-threaded programs on multi-cores.

ACKNOWLEDGMENT

This work was partially supported by the National Science Foundation under grants CNS-0746913 and CNS-0708788, and an equipment donation from Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF or Intel.

REFERENCES

[1] G. E. Suh, J. Lee, D. X. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the* 11^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[2] J. Newsome and D. Song, "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *Proceedings of the* 2005 *Network and Distributed Systems Symposium*, February 2005.

[3] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *Proceedings of the* 34^{th} *International Symposium on Computer Architecture*, June 2007.

[4] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *Proceedings of the* 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 304–316.

[5] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the C programming language," in *ASPLOS XIII: Proceedings of the* 13^{th} *international conference on Architectural support for programming languages and operating systems*, 2008, pp. 103–114.

[6] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proceedings of the* 22^{nd} *International Conference on Automated Software Engineering*, 2007.

[7] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, "iWatcher: Efficient architectural support for software debugging," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[8] J. Joao, O. Mutlu, and Y. Patt, "Flexible reference-counting-based hardware acceleration for garbage collection," in *Proceedings of the* 36^{th} *International Symposium on Computer Architecture*, 2009.

[9] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Proceedings of the* 40^{th} *Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[10] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the* 39^{th} *International Symposium on Microarchitecture*, 2006.

[11] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. Gibbons, T. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *Proceedings of the* 35^{th} *International Symposium on Computer Architecture*, June 2008.

[12] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "MemTracker: Efficient and programmable support for memory access monitoring and debugging," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 273–284.

[13] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "FlexiTaint: A programmable accelerator for dynamic taint propagation," in *Proceedings of the* 14^{th} *International Symposium on High Performance Computer Architecture*, 2008.

[14] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors in C and C++ programs," in *Proceedings of the Winter 1992 USENIX Conference*, 1992, p. 125138.

[15] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proceedings of the* 32^{th} *International Symposium on Microarchitecture*, November 1999.

[16] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the* 37^{th} *International Conference on Microarchitecture*, December 2004.

[17] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of the* 11^{th} *Annual Network and Distributed System Security Symposium*, 2004.

[18] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Proceeding of the* 28^{th} *International Conference on Software Engineering*, May 2006.

[19] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[20] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008, pp. 263–277.

[21] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 122–135.

[22] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proceedings of the* $5th$ *Annual IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[23] H. Kannan, "Ordering decoupled metadata accesses in multiprocessors," in *ACM/IEEE* 42^{nd} *International Symposium on Microarchitecture (MICRO-42)*, December 2009.

[24] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc, "GRLIB IP Core User's Manual," 2008.

[25] I. Kuon and J. Rose, "Area and delay trade-offs in the circuit and architecture design of FPGAs," in *Proceedings of the 2008 ACM/SIGDA* 16^{th} *International Symposium on Field Programmable Gate Arrays*, 2008.

[26] Xilinx, "Virtex-5 power spreadsheet," 2010.

[27] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *Annual IEEE International Workshop on Workload Characterization*, 2001.

[28] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The chimaera reconfigurable functional unit," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 2, 2004.

[29] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the* 27^{th} *Annual International Symposium on Microarchitecture*, 1994.

[30] H.-S. Kim, A. K. Somani, and A. Tyagi, "A reconfigurable multifunction computing cache architecture," in *Proceedings of the 8th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2000.

[31] R. D. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[32] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh, "INDRA: An integrated framework for dependable and revivable architectures using multicore processors," in *Proceedings of the* 33^{rd} *International Symposium on Computer Architecture*, 2006.

[33] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta, "Dynamic information flow tracking on multicores," in *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[34] S. Mysore, B. Agrawal, N. Srivastava, S.-C. Lin, K. Banerjee, and T. Sherwood, "Introspective 3D chips," in *Proceedings of the* 12^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[35] B. Quinton and S. Wilton, "Post-silicon debug using programmable logic cores," in *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology*, 2005.

[36] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 7–12.

[37] I. Wagner, V. Bertacco, and T. Austin, "Shielding against design flaws with field repairable control logic," in *Proceedings of the 43rd ACM/IEEE Design Automation Conference*, 2006.